AD-A093 261 VIRGINIA POLYTECHNIC INST AND STATE UNIV WASHINGTON --ETC F/G 9/2
FILES IN AN INTERACTIVE ENVIRONMENT.(U)
APR 80 R J ORGASS AFOSR-79-0021
UNCLASSIFIED VPI/SU-TM-80-3 AFOSR-TR-80-1289 NL

END
DATE FILMED
2-81
DTIC

EXTENSION DIVISION

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE
GRADUATE PROGRAM IN NORTHERN VIRGINIA

P. O. Box 17186
Washington, D. C. 20041
(703) 471-4600

FILES IN AN INTERACTIVE ENVIRONMENT*

Richard J. Orgass

Technical Memorandum No. 80-3

April 1, 1980

SUMMARY

A well designed file system can significantly simplify the design of and increase the reliability of interactive programs. A summary of the specifications of a convenient file system that is substantially independent of the host operating system and some experiences using the system are described.

Key Words: file systems, interactive systems

---

* Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, under Grant No. AFOSR-79-0021. The United State Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

REPORT DOCUMENTATION PAGE (19)		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (18) AFOSR-TR- 80-1289	2. GOVT ACCESSION NO. AD-A093 261	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) FILES IN AN INTERACTIVE ENVIRONMENT.		5. TYPE OF REPORT & PERIOD COVERED Interim
	(14)	6. PERFORMING ORG. REPORT NUMBER VPI/SU-TM-80-3
7. AUTHOR(s) (10) Richard J./Orgass	(15)	8. CONTRACT OR GRANT NUMBER(s) AFOSR-79-0021
9. PERFORMING ORGANIZATION NAME AND ADDRESS Dept. of Computer Science Virginia Polytechnic Institute and State University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F (16) 2304/A5 (17)
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Rearch /NM Bolling AFB, Washington, D. C . 20332		12. REPORT DATE (11) 1 April 1980
		13. NUMBER OF PAGES 9
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (12) 12		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

file systems, interactive systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A well designed file system can significantly simplify the design of and increase the reliability of interactive programs. A summary of the specifications of a convenient file system that is substantially independent of the host operating system and some experiences using the system are described.

411 731

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FILES IN AN INTERACTIVE ENVIRONMENT

Introduction

This writer has observed both in his own work and in the work of colleagues that one's willingness to provide a convenient interactive program is severely limited by the capabilities of the file system of the host computing system. If it is straightforward to provide many user conveniences, one is happy to do so but if complicated and tedious coding is required there is a strong temptation to cut corners at the expense of flexibility and convenience for users of the program.

For example, when using TOPS-10 which provides a very convenient file system with a few limitations, one tends to write programs that exploit the existing file system and leaves the limitations to be dealt with by the user. In contrast, in the VM/CMS environment, almost all file operations are reasonably difficult and there is a strong tendancy to leave almost everything to the user of a program and the result is programs that are very hard to work with. This difference in file systems was forcefully brought to the writer's attention when he moved a number of large interactive programs that were designed for use with TOPS-10 to VM/CMS.

When these programs were first moved, they were modified to simply bring them into operation in the new environment. When this task was completed, two observations were made: (1) A fair body of terribly uninteresting tedious code had been introduced into the programs. (2) It was very much more difficult to use the programs -- there were many more details left to the user and many small errors caused the loss of a significant amount of work during a terminal session. In addition, the need to deal with these details interfered with the work for which the programs were designed.

The second observation motivated yet additional program modifications to simplify the use of the programs and the result was programs that were so complicated that they were neither reliable nor amenable to further modification. Yet, the user interface still had annoying properties.

This state of affairs motivated the construction of the file system described here. The system was designed by implementing several prototype systems and using these systems to isolate further problems and to remove unnecessary attributes of files that had been introduced because they appeared to be useful. The final version provides for the easy implementation of very convenient programs with essentially no coding to deal with the files.

Assumptions

It is assumed that the host operating system provides some form of a directory of a user's files and that this directory associates a file name with each file in the directory. Input/output devices (e.g., terminals, line printers, readers, tape drives, etc.) have distinguished file names associated with them. The directory may also contain detailed information about the organization of the file which is needed to read the file; this information should be of no concern to either the programmer or the user of a program. Lastly, the file name that appears in a directory is also the name by which the file is known to a running program. [This requirement does not impose any restriction: If different files are to be processed in different executions of a program, these file names can be read from the terminal or from some other device.]

A terminal serves as both an input file and an output file and in this respect it is indistinguishable from other files. However, terminal input is typed by a user and output is often written to CRT terminals and, therefore, effective use of an interactive program requires additional capabilities for terminal files to make a user's task easier. For example, it should be possible to record user input in a disk file so that on subsequent executions of a program this earlier input can be read from the file followed by new user created inputs without requiring program modification or complicated terminal dialogs. Similarly, it should be possible to record terminal output in a disk file so that a record of the output is available for later review using an editor or for further processing.

There are unpublished studies which indicate that using tab characters can increase the effective data rate of a terminal by 50% and, in some file organizations, reduce the required storage space by 50%. Therefore, tabs are to be be used for input/output whenever this is compatible with the device. A programmer need not deal with these tabs at all: They are to be expanded into blanks on input and automatically inserted on output. In this kind of an environment, one step in the login procedure is to inform the operating system of the presence or absence of hardware tabs on the terminal.

File Objects

For a programmer, a file object is an abstract data type with procedure and data attributes. The data attributes of a file object include the name of the external file, the record of the external file that is currently being read or written and the position in this record of the next character to be read or written. The procedure attributes include procedures to open and close the external file, to transmit a record between the one record buffer and the external file, to test for end-of-file, and, possibly, procedures to transmit data to various types to or from the one record buffer. Such a file object is used as follows:

When a file object is created, the name of the external file is passed as a parameter. Next, the external file is opened and the one record buffer is created. After this, records are transmitted between the external file and the program via the one record buffer. Finally, after data transfer is completed, the external file is closed. If the file should be opened again, reading or writing begins with the first record of the file and in the case of an output file the original contents are discarded.

This view of file objects is essentially the view adopted in SIMULA-67[1]. In the following sections, extensions to this view which include additional capabilities and error recovery are described.

Input Files

While using an input file object to read an external file a number of user or programmer errors may occur and run time error recovery to include at least the following is required.

If the file name that is passed to an input file object when it is created is not the name of an existing file then a corrective message is printed on the terminal and the user is asked to provide the correct file name, possibly after consulting the directory.

If there is an attempt to open an already open file, a corrective message is printed on the terminal and the user is given the option of continuing or terminating execution.

If there is an attempt to read from an external file when the file is closed, a corrective message is printed on the terminal and the user is given the option of either opening the file and continuing execution or terminating execution.

During an input operation, if the next record of the external file is longer than the provided input buffer, a corrective message is printed on the terminal and the user is given the option of extending the input buffer or terminating execution.

If there is an attempt to close an already closed file an advisory message is printed on the terminal and execution continues.

The action taken on an attempt to read past an end-of-file is determined by the value of a boolean attribute, divert, of an input file object. If divert is false, then a corrective message is printed on the terminal and the user is given the option of rereading the file or terminating execution. On the other hand, if divert is true, then an advisory message is printed on the terminal and subsequent reads from this file are read from the terminal without providing an end-of-file indication to the program. Note that if divert is true, then terminal input becomes an automatic extension of any input file.

In some cases, it is desirable to reproduce on the terminal input read from an external file other than the terminal as records are read from the file. This is particularly useful when the data read from a file creates the environment in which a user interacts with the program. The boolean attribute echo of an input file object provides this capability. If echo has the value true, then the above copying to the terminal occurs and if it is false this copying does not occur.

Input file objects have an output file object log as an attribute. All lines read from the external input file are copied to the external file associated with the log. If log is the empty output file object, then this copying does not occur. This attribute of an input file object makes it straightforward to create files containing terminal input.

Finally, some small details that make input files associated with the terminal much easier to work with during program execution. If a user simply enters carriage return when input is expected, this input is interpreted as an empty line of input and not as an end-of-file. An input line whose first character is a control character serves as an end-of-file from the terminal. In addition, a second control character appearing as the first character of an input line terminates execution of a program. Implementing this convention in the CMS environment, removed about 30% of all user input errors and substantially simplified many programs.

Input Streams

The record oriented input file objects described above are adequate for many applications but they quickly break down when one wishes to consider a file as a sequence of characters. While it is certainly possible to write code to read a file as a sequence of characters, one then has two kinds of objects that are strings of characters: input files interpreted as character strings and program variables that are strings of characters (text objects). If a program processing these two kinds of character strings is obliged to distinguish between them, the program quickly becomes cluttered with a wide variety of terribly uninteresting and complicated code which provides many opportunities for small errors. Both the reliability and comprehensibility of such code is severely compromised because irrelevant details have not been suppressed by the programming environment.

This problem disappears if the environment provides an object that may be interpreted as a string of characters independent of the source of the characters. One wishes to distinguish between the source of the characters when the object is created but when working with the object, there is no need to know the source of the characters. Objects of type stream appear to meet this requirement quite well.

When a stream is created, two parameters are passed: a text object and a boolean. The second parameter provides an interpretation of the first parameter as follows: If the second parameter is true, then the value of the first parameter is interpreted as the name of a file and this file is the source of characters. On the other hand, if the value of the second parameter is false, then the first parameter itself is the source of characters.

Stream objects have some of the attributes of text objects and some of the attributes of input file objects. Since a stream may be associated with an external file, it must be opened and closed as an input file object and it must be possible to test for end-of-file. Characters are read from a stream using the primitives appropriate to a text object since these are closer to the concept of the object. This implementation of a stream provides considerable simplification of programs but there are some obvious generalizations.

One quickly discovers that a stream is much more convenient than an input file object but one quickly creates programs where there is a need to skip to the next input record. For example, if a stream is the input to a parser for program text, one might want to skip to the next record when a comment symbol is encountered. Therefore, a primitive to read the next record is desirable. Of course, if the source of characters is a text object, end-of-file becomes true after the first read of the next record.

If the source of characters for a stream is an external file, the attributes echo and log of input file objects have the same meaning; if the source of characters is a text object, they have no effect. The divert attribute of input file objects has the same interpretation for streams independent of the source of characters.

If a file is viewed as a source of characters, another generalization immediately suggests itself. At some point in a file, one might wish to include all of the characters in another file and then continue reading the original file. Therefore, if a specific character, called the indirect file character, appears in the first column of an input record, the remainder of the record is interpreted as the name of a file and input is read from this file until end-of-file is encountered. At this point, input is again taken from the first file. Any number of indirect files may be in use at any time subject only to the host system restriction on the maximum number of open files.

Output Files

While using an output file object to write to an external file a number of user or programmer errors may occur and run time error recovery to include at least the following is required.

If the file name that is passed to an output file object when it is created is such that it is not possible to write to this file then a corrective message is printed on the terminal and the user is asked to provide the correct file name, possibly after consulting the directory. This can occur, for example, if the file name refers to a directory to which the program has read-only access or if there is no space available for the file.

Error recovery for many attributes of an output file object is essentially the same as for an input file object. However, there is one important difference. If there is an attempt to write an output line that is longer than the buffer associated with the output file object, the output line is written as a sequence of output records and no error indication is given.

Output file objects also have an echo attribute which controls the behavior of the object as follows. If the external file associated with the output file object is not the terminal, the echo attribute is meaningful: If echo is true, output records are written to the external file and to the terminal. If echo is false, this terminal output is not written. This substantially simplifies writing programs that record output for further processing or later examination while also providing a copy on the terminal to guide the user.

There is one additional detail of output file objects that significantly simplifies programs. When output is written to a file, tab characters are automatically inserted to replace blanks. This provides for substantially smaller disk files and for higher speed output on terminals. It is the responsibility of the host operating system to expand tabs into blanks when writing to a terminal or other device that lacks hardware tabs. Again, this removes a significant amount of annoying details from programs while providing for more efficient use of storage space.

Implementation

The file primitives described here have been implemented using SIMULA-67 in the VM/CMS operating system. The code consists of approximately 1500 lines of SIMULA of which about 40% is documentation. This code is written in the Common Base definition of SIMULA and should also function correctly in the DEC-10 SIMULA implementation.

In the VM/CMS environment, this code is supported by approximately 500 lines of assembly code and approximately 800 additional lines of SIMULA code. All of this program text strongly depends on the VM/CMS environment. The assembly code provides direct terminal input/output bypassing the SIMULA run time system code which is inadequate in this environment and the SIMULA code makes it possible to associate CMS DD names with CMS files at run time and to suppress the details associated with using DD names.

In addition, this code uses modifications to VM written by the University Computing Center to support tabs on terminals as described.

In the DEC-10 environment, approximately 50 lines of system dependent SIMULA code would be required to bring the file system into correct operation.

Approximately one man year was devoted to the design and implementation of this file system. Since this writer was learning to use VM/CMS at the same time, some of this effort was simply education in the obscure details of the system.

Evaluation

The file system described here has been incorporated into two programs: an APL implementation and an interactive program verification system. Both of these programs were designed and brought into operation in DEC-10 SIMULA and then transferred to VM/CMS and IBM SIMULA.

At the beginning of this program transfer, the programs were modified incrementally to deal with a variety of problems that were encountered when using them in the VM/CMS environment.

The APL implementation uses disk files in a very limited way but makes heavy demands on the terminal interface. After the file system was implemented, it was incorporated into the APL implementation and approximately 800 lines of very uninteresting and fragile code were removed. The revised version is substantially simpler and can be modified easily. In addition, error recovery appears to be complete.

The interactive program verification system makes heavy use of disk files and the echoing and logging functions associated with files. Before the file system was implemented, approximately 1100 lines of code had been introduced to provide marginally acceptable behavior in the VM/CMS environment. All of this code was removed when the file system was implemented. In addition, approximately 500 lines of code that was created to provide some of the echoing and logging capabilities in the DEC-10 environment was removed. Again, the resulting program is much cleaner and simpler and substantially more reliable. In addition, it is now very easy to make modifications in the user interface without introducing complicated code.

If these two programs were the only application of this file system, the development effort would be justified. However, the file system has also been used to create a variety of other utility programs that provide some locally useful function and some programs that were created before the file system was implemented have been revised to use the file system. The revised programs are substantially smaller than previous versions and the creation of additional utilities is now a trivial task. Work that once consumed several days can now be done in an hour or two.

There are other advantages associated with using this file system that are much more difficult to document precisely. Since it is very easy to provide a convenient terminal interface, one automatically writes programs in this way. The same primitives are used in a variety of programs and this substantially simplifies program documentation and user training. The code that depends on specific properties of the operating system is isolated and well defined and, therefore, moving programs to another host system is very much easier. Only a limited number of specific procedures must be rewritten and the external behavior of programs is essentially unchanged. The only change is a different syntax for file names.

In summary, this file system has been found to be an extremely useful tool for the development of interactive programs with minimal attention to the details of input/output. The file system provides a very comfortable, convenient environment for program creation.

Acknowledgement

It is a pleasure to thank Richard W. Critz, Jr. for writing the assembly code that supports this file system in VM/CMS and for making many constructive suggestions about the design of the system.

Reference

1. O.-J. Dahl, B. Myhrhaug and K. Nygaard, *The SIMULA-67 Common Base Language*, Publication No. S-2, Norwegian Computing Center, Oslo, May 1978.

DATE
ILMED

- 8